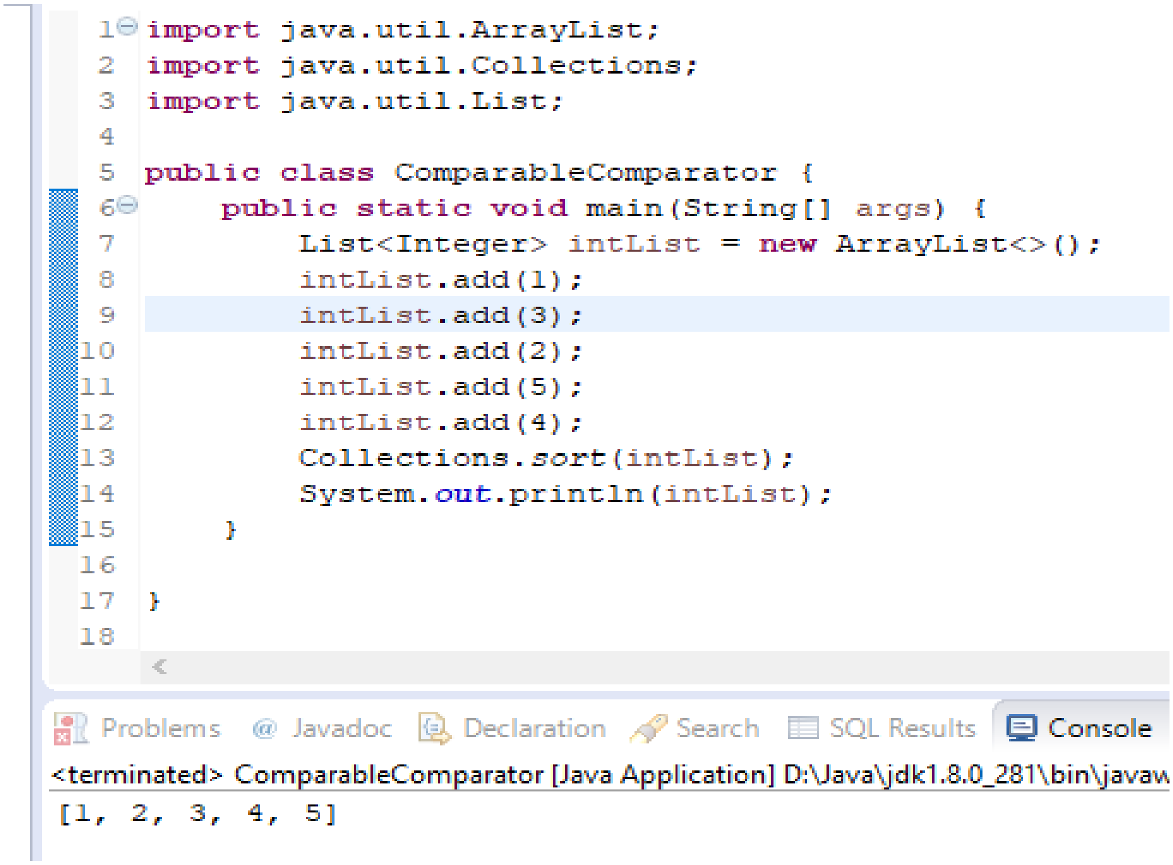
Comparable and Comparator

Comparable: *Comparable* is an interface defining a strategy of comparing an object with other objects of the same type. This is called the class's “natural ordering”. The sorting order is decided by the return value of the *compareTo()* method.
The *Comparable* interface is a good choice when used for defining the default ordering or, in other words, if it's the main way of comparing objects.

Collection.sort()

Collection.sort()method is present in java.util.Collections class. It is used to sort the elements present in the specified **list** of Collection in ascending order.
It works similar to **java.util.Arrays.sort()** method but it is better then as it can sort the elements of Array as well as linked list, queue and many more present in it.

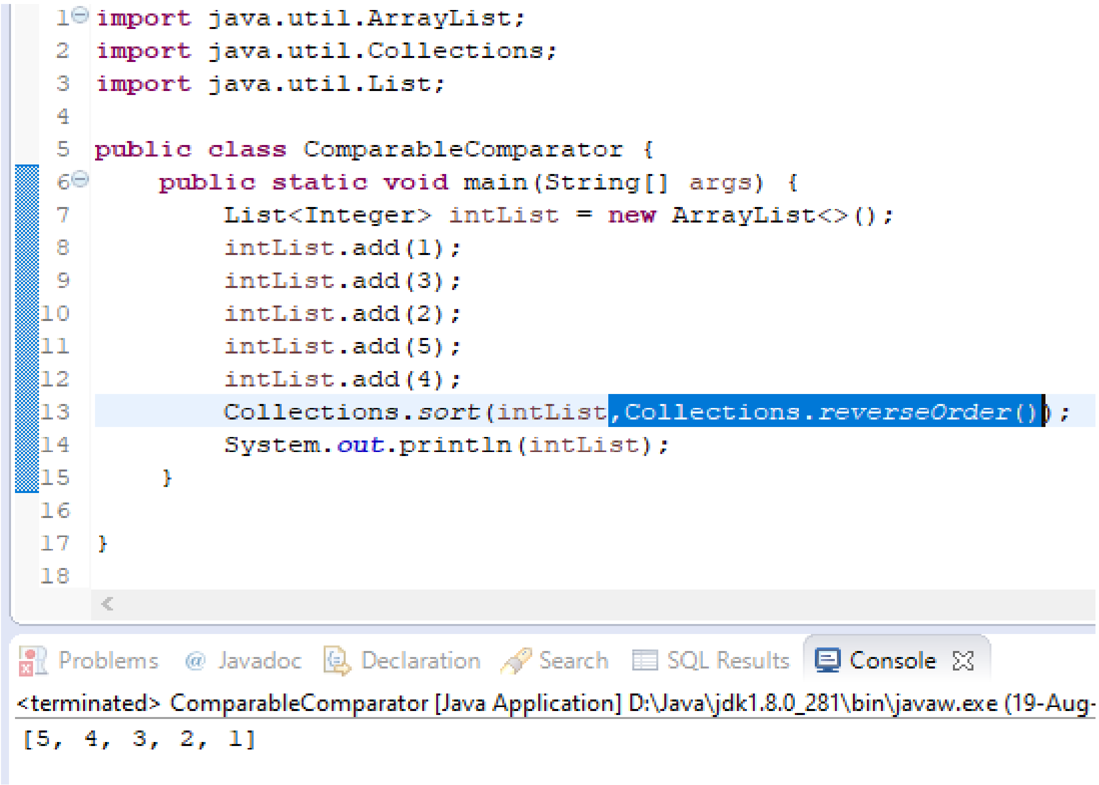
Sorting in Ascending Order:



```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ComparableComparator {
    public static void main(String[] args) {
        List<Integer> intList = new ArrayList<>();
        intList.add(1);
        intList.add(3);
        intList.add(2);
        intList.add(5);
        intList.add(4);
        Collections.sort(intList);
        System.out.println(intList);
    }

}

```

```
<terminated> ComparableComparator [Java Application] D:\Java\jdk1.8.0_281\bin\javaw
[1, 2, 3, 4, 5]
```

Sorting in Descending Order:



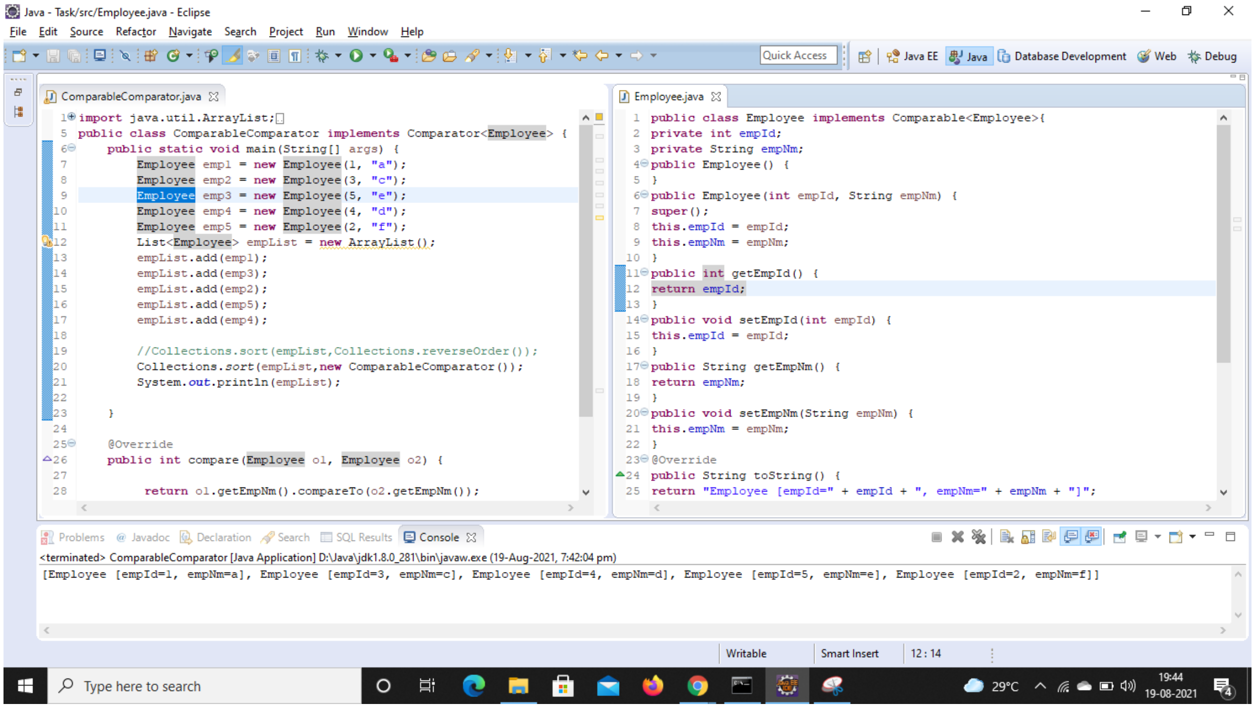
```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ComparableComparator {
    public static void main(String[] args) {
        List<Integer> intList = new ArrayList<>();
        intList.add(1);
        intList.add(3);
        intList.add(2);
        intList.add(5);
        intList.add(4);
        Collections.sort(intList,Collections.reverseOrder());
        System.out.println(intList);
    }

}

```

```
<terminated> ComparableComparator [Java Application] D:\Java\jdk1.8.0_281\bin\javaw.exe (19-Aug-
[5, 4, 3, 2, 1]
```

Comparator: The *Comparator* interface defines a *compare(arg1, arg2)* method with two arguments that represent compared objects and works similarly to the *Comparable.compareTo()* method.
To create a *Comparator,* we have to implement the *Comparator* interface.

1. Both are interfaces; they have methods used for comparing objects.
2. Comparable can be implemented within the same class - (only one variable comparison can be comparison)
3. Comparator we no need to implement
   The comparison logic within the same class - different comparison logic i can.
4. To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if you sorting needs to be done on attributes of different objects, then use Comparator in Java.

Example:


Java - Task/src/Employee.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java EE Java Database Development Web Debug
ComparableComparator.java
Employee.java
Problems Javadoc Declaration Search SQL Results Console
Writable Smart Insert 12 : 14
Type here to search
29°C
19:44
19-08-2021

```java
import java.util.ArrayList;
public class ComparableComparator implements Comparator<Employee> {
    public static void main(String[] args) {
        Employee emp1 = new Employee(1, "a");
        Employee emp2 = new Employee(3, "c");
        Employee emp3 = new Employee(5, "e");
        Employee emp4 = new Employee(4, "d");
        Employee emp5 = new Employee(2, "f");
        List<Employee> empList = new ArrayList();
        empList.add(emp1);
        empList.add(emp3);
        empList.add(emp2);
        empList.add(emp5);
        empList.add(emp4);

        //Collections.sort(empList,Collections.reverseOrder());
        Collections.sort(empList,new ComparableComparator());
        System.out.println(empList);

    }

    @Override
    public int compare(Employee o1, Employee o2) {

         return o1.getEmpNm().compareTo(o2.getEmpNm());
```

```java
public class Employee implements Comparable<Employee>{
private int empId;
private String empNm;
public Employee() {
}
public Employee(int empId, String empNm) {
super();
this.empId = empId;
this.empNm = empNm;
}
public int getEmpId() {
return empId;
}
public void setEmpId(int empId) {
this.empId = empId;
}
public String getEmpNm() {
return empNm;
}
public void setEmpNm(String empNm) {
this.empNm = empNm;
}
@Override
public String toString() {
return "Employee [empId=" + empId + ", empNm=" + empNm + "]";
```

```
<terminated> ComparableComparator [Java Application] D:\Java\jdk1.8.0_281\bin\javaw.exe (19-Aug-2021, 7:42:04 pm)
[Employee [empId=1, empNm=a], Employee [empId=3, empNm=c], Employee [empId=4, empNm=d], Employee [empId=5, empNm=e], Employee [empId=2, empNm=f]]
```

You should not implement Collections.sort(); this method is built-in to Java. Call that method without supplying a Comparator to sort by the *natural order*, if it's Comparable. Else, supply a Comparator to sort the Comparator's way.

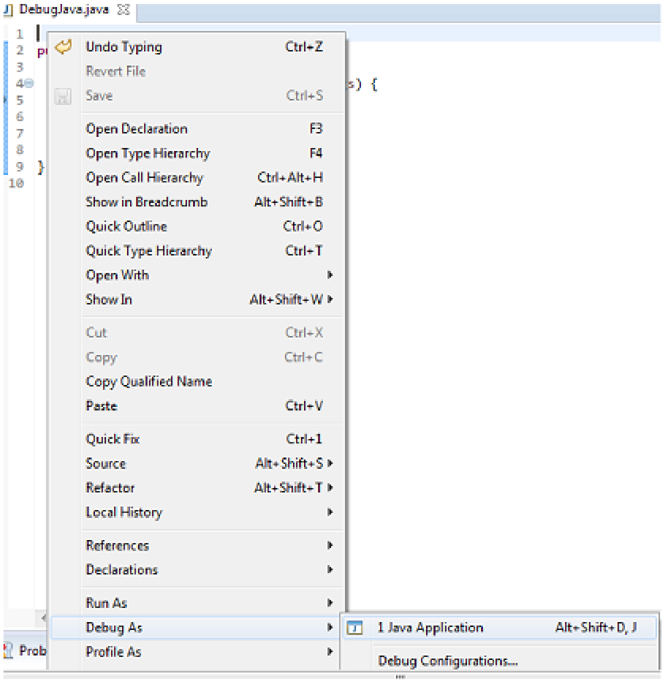
You should have the class implement Comparable and provide a compareTo method if the class has a *natural ordering*, as indicated in the Comparable javadocs. An example would be for Integer and Double, which certainly have a natural mathematical ordering.

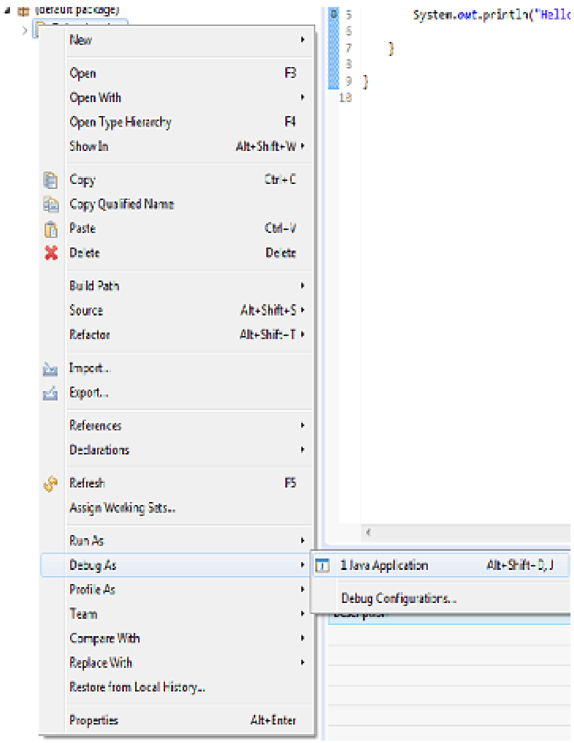
You should create a class that implements Comparator when you cannot make the class of the object to sort Comparable or when you want to present an ordering that is an alternative to the natural ordering, or when you want to impose an order when there is no natural ordering. An example would be to reverse the natural order (say, sort descending, from largest to smallest). Another example would be a data object with multiple fields, that you want to be sortable on multiple fields, e.g. a Person object with firstName and lastName: you could have a Comparator that sorts on lastName first, and another that sorts on firstName first.

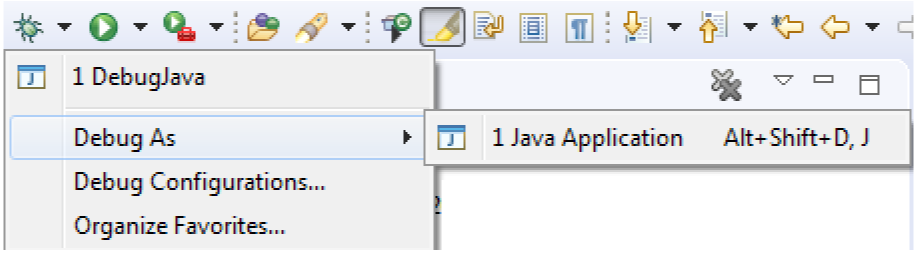
How to Debug the Java Code

1. Launching and Debugging a Java program

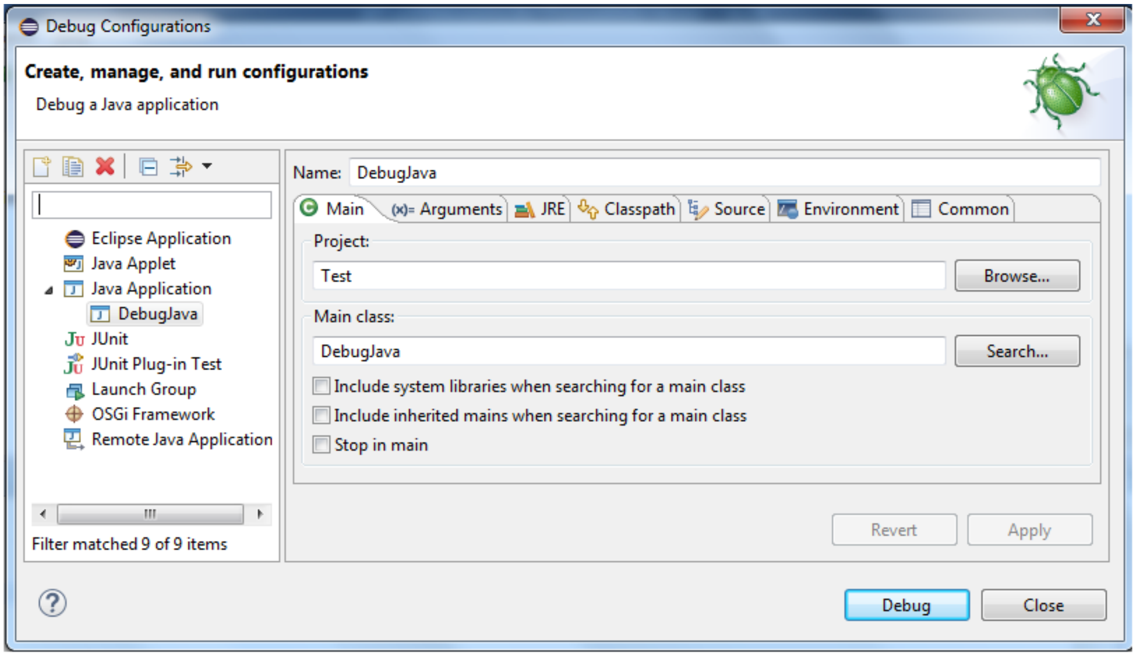
A Java program can be debugged simply by right clicking on the Java editor class file from Package explorer. Select **Debug As** → **Java Application** or use the shortcut **Alt + Shift + D, J** instead.







Either actions mentioned above creates a new **Debug Launch Configuration** and uses it to start the Java application.



Name: DebugJava

Project:
Test

Main class:
DebugJava

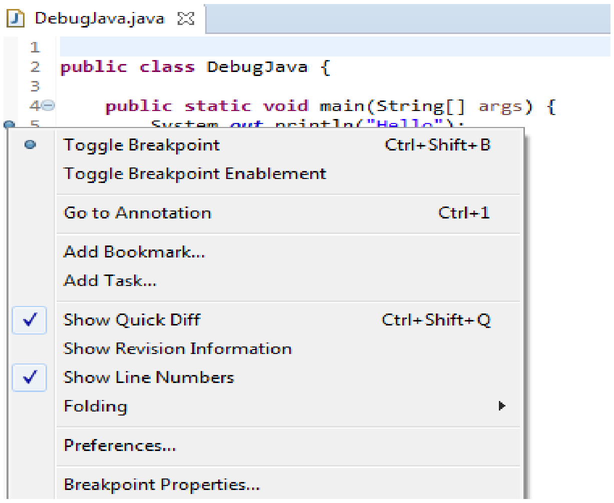
☐ Include system libraries when searching for a main class
☐ Include inherited mains when searching for a main class
☐ Stop in main

In most cases, users can edit and save the code while debugging without restarting the program.This works with the support of **HCR** (Hot Code Replacement), which has been specifically added as a standard Java technique to facilitate experimental development and to foster iterative trial-and-error coding.

2. Breakpoints

A breakpoint is a signal that tells the debugger to temporarily suspend execution of your program at a certain point in the code.

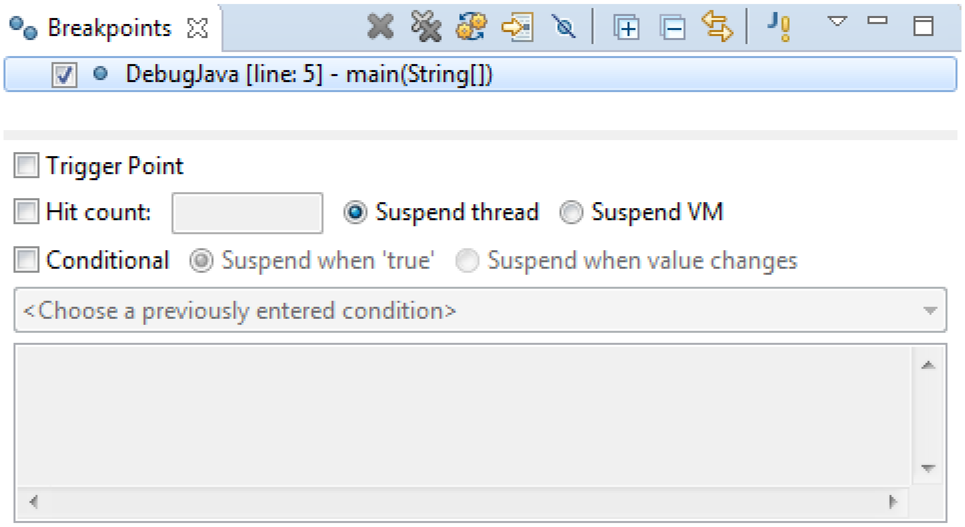
To define a breakpoint in your source code, right-click in the left margin in the Java editor and select *Toggle Breakpoint*. Alternatively, you can double-click on this position.


DebugJava.java

```

public class DebugJava {

    public static void main(String[] args) {
        System.out.println("Hello");
```

Toggle Breakpoint Ctrl+Shift+B
Toggle Breakpoint Enablement
Go to Annotation Ctrl+1
Add Bookmark...
Add Task...
✓ Show Quick Diff Ctrl+Shift+Q
Show Revision Information
✓ Show Line Numbers
Folding
Preferences...
Breakpoint Properties...

The *Breakpoints* view allows you to delete and deactivate Breakpoints and modify their properties.



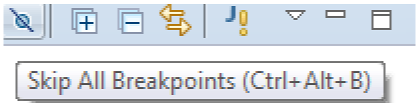
☑ DebugJava [line: 5] - main(String[])

□ Trigger Point

□ Hit count: ● Suspend thread ○ Suspend VM

□ Conditional ● Suspend when 'true' ○ Suspend when value changes

<Choose a previously entered condition>

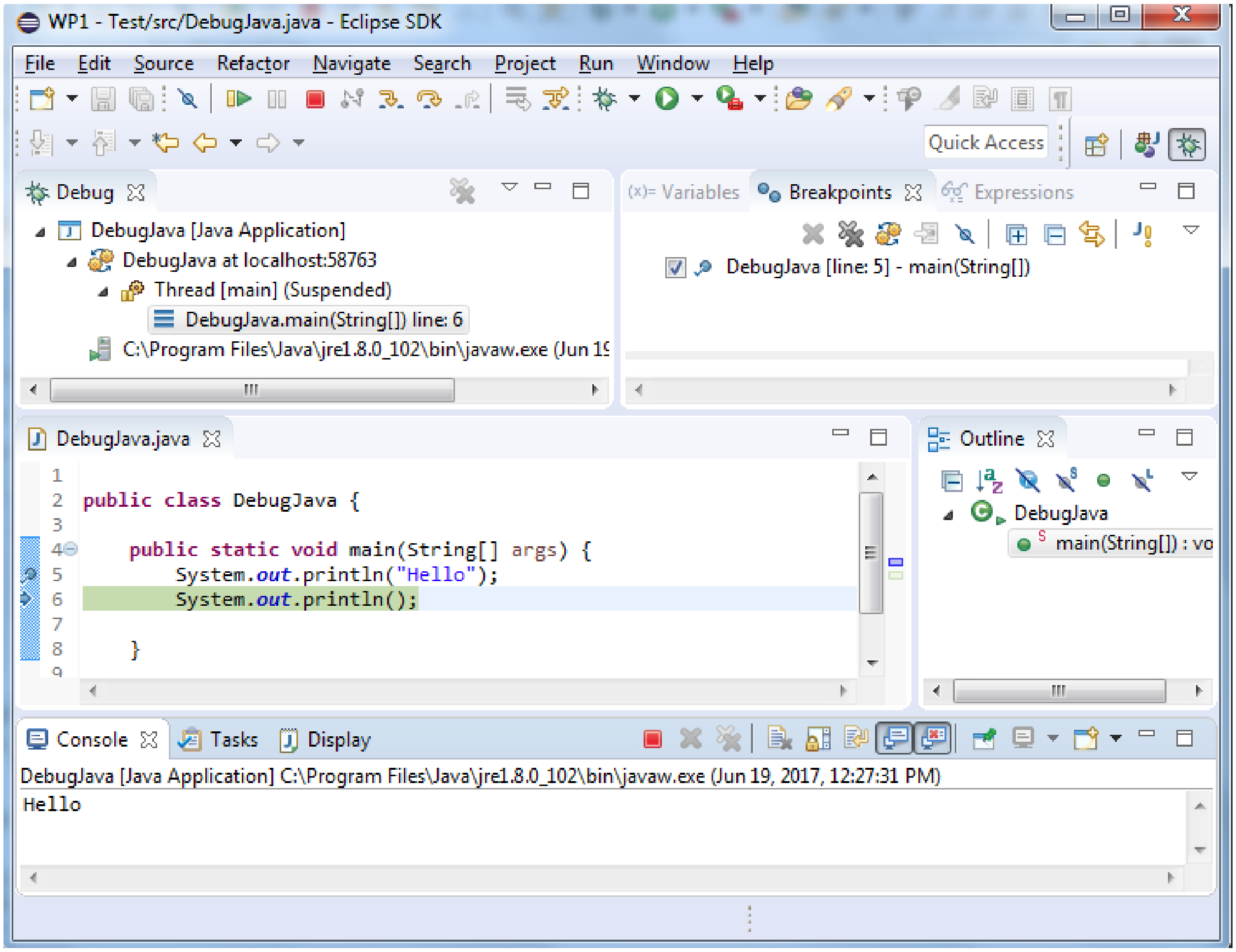


All breakpoints can be enabled/disabled using **Skip All Breakpoints**. Breakpoints can also be imported/exported to and from a workspace.

3. Debug Perspective

The debug perspective offers additional views that can be used to troubleshoot an application like Breakpoints, Variables, Debug, Console etc. When a Java program is started in the debug mode, users are prompted to switch to the debug perspective.

- **Debug view** – Visualizes call stack and provides operations on that.
- **Breakpoints view** – Shows all the breakpoints.
- **Variables/Expression view** – Shows the declared variables and their values. Press **Ctrl+Shift+d** or **Ctrl+Shift+i** on a selected variable or expression to show its value. You can also add a permanent watch on an expression/variable that will then be shown in the *Expressions view* when debugging is on.
- **Display view** – Allows to Inspect the value of a variable, expression or selected text during debugging.
- **Console view** – Program output is shown here.



DebugJava [Java Application]
DebugJava at localhost:58763
Thread [main] (Suspended)
DebugJava.main(String[]) line: 6
C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Jun 19

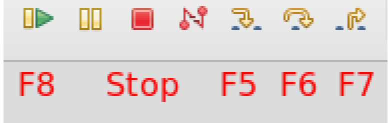
☑ DebugJava [line: 5] - main(String[])

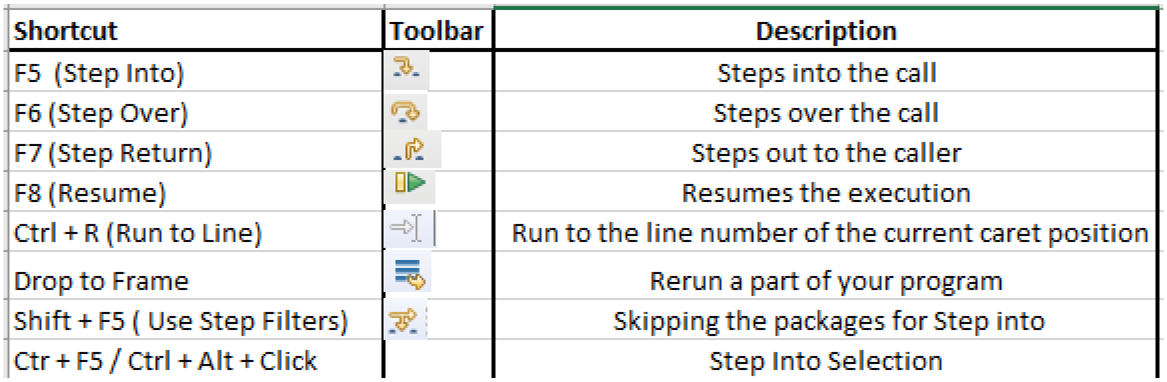
```

public class DebugJava {

    public static void main(String[] args) {
        System.out.println("Hello");
        System.out.println();

    }
```

```
DebugJava [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Jun 19, 2017, 12:27:31 PM)
Hello
```

4.Stepping commands

The Eclipse Platform helps developers debug by providing buttons in the toolbar and key binding shortcuts to control program execution.





Shortcut	Toolbar	Description
F5 (Step Into)	⤵	Steps into the call
F6 (Step Over)	↷	Steps over the call
F7 (Step Return)	⤴	Steps out to the caller
F8 (Resume)	▷	Resumes the execution
Ctrl + R (Run to Line)	⇥	Run to the line number of the current caret position
Drop to Frame	≡	Rerun a part of your program
Shift + F5 (Use Step Filters)	⤵	Skipping the packages for Step into
Ctr + F5 / Ctrl + Alt + Click		Step Into Selection

For more information about debugging visit: Eclipse Stepping Commands Help